C++2MPI/PGMT Integration

by

Roger Hillson

Revised 23 July 2002

1. Introduction

In PGM, the basic data structure is a hierarchical entity called a PGM *family*. PGM families are strongly-typed. Each PGM family has some specified *base-type* or *leaf-node* type. Although the PGM specification is language-independent, PGM is always implemented in some specific high-order language (HOL). A PGM user can define his own family base-types, and/or he can use predefined base-types corresponding to the predefined types in the PGM high-order language (HOL). PGMT proper is implemented in C++. In the PGMT environment, a user can define a base-type for a PGM family by specifying a C++ class. A user can also define a family whose base-type is a predefined C++ arithmetic datatype: for example, *int, float, double,* or *long*.

PGMT includes a library of Middleware. The Middleware library includes message-passing functions which can be called by functions in the Graph Class Library (GCL) or PEP (PGMT Execution Program). All of the Middleware communication functions use calls to MPI 1.1. (Message Passing Interface 1.1) function libraries to send and receive data. The MPI function calls are hidden from the GCL and PEP function calls. In order to call an MPI function to send or receive data, the *MPI datatype* for the data must be specified. (Note: in accordance with the MPI 1.1. specification, "datatype" is spelled as a single word.) To specify the MPI datatype of the data being sent or received, it is necessary to pass an MPI *handle* as an argument to the MPI function being called. An MPI handle is an opaque object of type *MPI_Datatype*. In order to use MPI to pass instances of PGM families i.e. PGM tokens containing families, there must be an MPI handle corresponding to the base-type of the PGM family.

PGMT is a tool which accepts one or more C++ classes as inputs, and then generates a function which will build the corresponding MPI datatype for the class. The function returns a handle to the MPI datatype. Thus C++2MPI permits a user to automatically create MPI types for sending and receiving instances of C++ classes.

2. MPI and PGM datatypes

There is a distinction between the MPI handle for a datatype, and the underlying datatype itself. An MPI datatype is basically a memory map, consisting of ordered pairs of MPI datatypes and their offsets in bytes from the beginning of a hypothetical send/receive buffer. By definition, an MPI datatype is a list of ordered pairs of the form (C type, buffer_offset). This list is called a *typemap*.

The MPI specification notes that, in an inefficient implementation of MPI, the typemap itself
could be passed as the MPI handle. In practice, the MPI handle (i.e. objects of type *MPI_Datatype*) is usually an index or pointer to the memory map, rather than the memory map

itself. In the free MPI implementation MPICH, MPI handles are simply enumerated types with some integer value. In the MPI implementation LAM, MPI handles are pointers to C structures (*structs*). From the standpoint of C++2MPI, the implementation details for objects of type *MPI_Datatype* do not matter, although it is helpful to understand the above considerations.

MPI provides predefined datatypes for common C datatypes. For example, *MPI_INT* is the predefined MPI handle for sending and receiving integers, and *MPI_FLOAT* is the predefined MPI handle for sending and receiving floats. PGMT defines base-types as objects of type *MW_Datatype*. In PGMT, *MW_Datatype* is type-defined as an object of type *MPI_Datatype*. By defining the type *MW_Datatype*, the use of MPI is hidden at the level of the PEP and GCL. Alternative middleware libraries could be written which do not use MPI for the communication calls, but the GCL and PEP middleware interfaces would remain unchanged.

3. C++2MPI assumptions

C++2MPI enables a PGMT user to automatically create an MPI datatype corresponding to some user-defined C++ class. Given the C++ class as an input, C++2MPI creates a function which, when compiled and called, returns an MPI handle (object) of type *MPI_Datatype*. In order to send or receive an instance of the user-defined class, the MPI handle must be passed as an argument to the MPI function being called e.g. *MPI_Send()* or *MPI_Receive()*. The current C++2MPI script also compiles and archives the functions which will create the MPI datatypes. The creation of an archive library creates some OS-specific dependencies; this issue will be addressed elsewhere.

(1) C++2MPI accepts as inputs:

(a) User class definitions for simple, derived, nested or templated classes. The class definitions must be properly delimited by #pragma statements. User-defined types containing pointers, static types, or unions are not supported.

(b) In the case of templated classes only, the user must also provide a type definition (typedef) for each distinct instantiation of the templated class.

(2) C++2MPI produces two output files: C++2MPI.h and C++2MPI.cpp. These files contain the prototypes and function bodies for the MPI datatype build functions. When called, the build functions return an object of type MPI_Datatype for each of the user-specified base-types. C++2MPI compiles the functions for building the user-defined MPI datatypes into an archive file `libC++2MPI.a`.

(3) To support the use of these datatype-building functions, C++2MPI also generates an auxiliary program `MPIBuildFunc.cpp`. This file defines the following pair of objects for *each* user-defined MPI datatype:

• a handle for each user-specified datatype. The handle is declared as an object of type MPI_Datatype.

• the address of the datatype-building function which returns the value of the corresponding datatype handle.

4. Run-time identification of middleware datatypes

Assume that C++2MPI has been run with one or more user-defined C++ classes as inputs. The functions for creating the user-defined types have been compiled into an archive. How does PGMT determine which MPI datatype to use when sending or receiving a family of some PGM base-type? The answer is that the PGMT GSF-to-C++ translator generates calls to the function *Machine::getMW_DataType(string)* (see Figure 1). The function returns a handle of type *MW_Datatype*, which is type-defined to *MPI_Datatype.*

The prototype for this function is:

typedef MPI_Datatype MW_Datatype;
MW_Datatype getMW_DataType(string arg);

A fragment of output from the PGMT Translator might be:

```
MW_Datatype MW_Int, MW_Float, MW_Complex;
MW_Int          = getMW_DataType("int");          // predefined type
MW_Float        = getMW_DataType("float");        // predefined type
MW_Complex      = getMW_DataType("complex");      // user-defined type
```

The key PGMT/C++2MPI integration issue is to build the function *Machine::getMW_DataType()*. This function must always return the proper handle for any *predefined* MPI datatype. It must also return the MPI datatype for any user-defined datatype. The function *Machine::getMW_DataType()* uses an associative map instantiated from the standard template library (STL). The key for each record is the name of the datatype, and the value returned is the address of the function which will build the datatype. This approach provides a uniform interface: for each predefined *or* user-defined datatype, a function must exist which will return an MPI handle to it.

4. Building getMW_DataType(string arg);

4.1 MPIBuildFunc.cpp C++2MPI generates an auxiliary file `MPIBuildFunc.cpp` which contains the definitions for an STL vector of structures. There is one structure defined for each user-defined type (i.e. class) provided by the user. Each structure has two components: the *name* of the user-defined type (i.e. class name) and the *address* of the function used to build the corresponding MPI datatype. For example, if a user defines a class named *complex,* the two components of the corresponding structure will be:

(1) *string(“complex”)*;
(2) *&build_complex_MPI_datatype*;

The first time the function *build_complex_MPI_datatype()* is called it builds the MPI datatype for the user-defined class complex, and returns an MPI handle of type *MPI_Datatype*. On subsequent calls to *build_complex_MPI_datatype()*, the function returns the correct value of the MPI handle without rebuilding the MPI datatype.

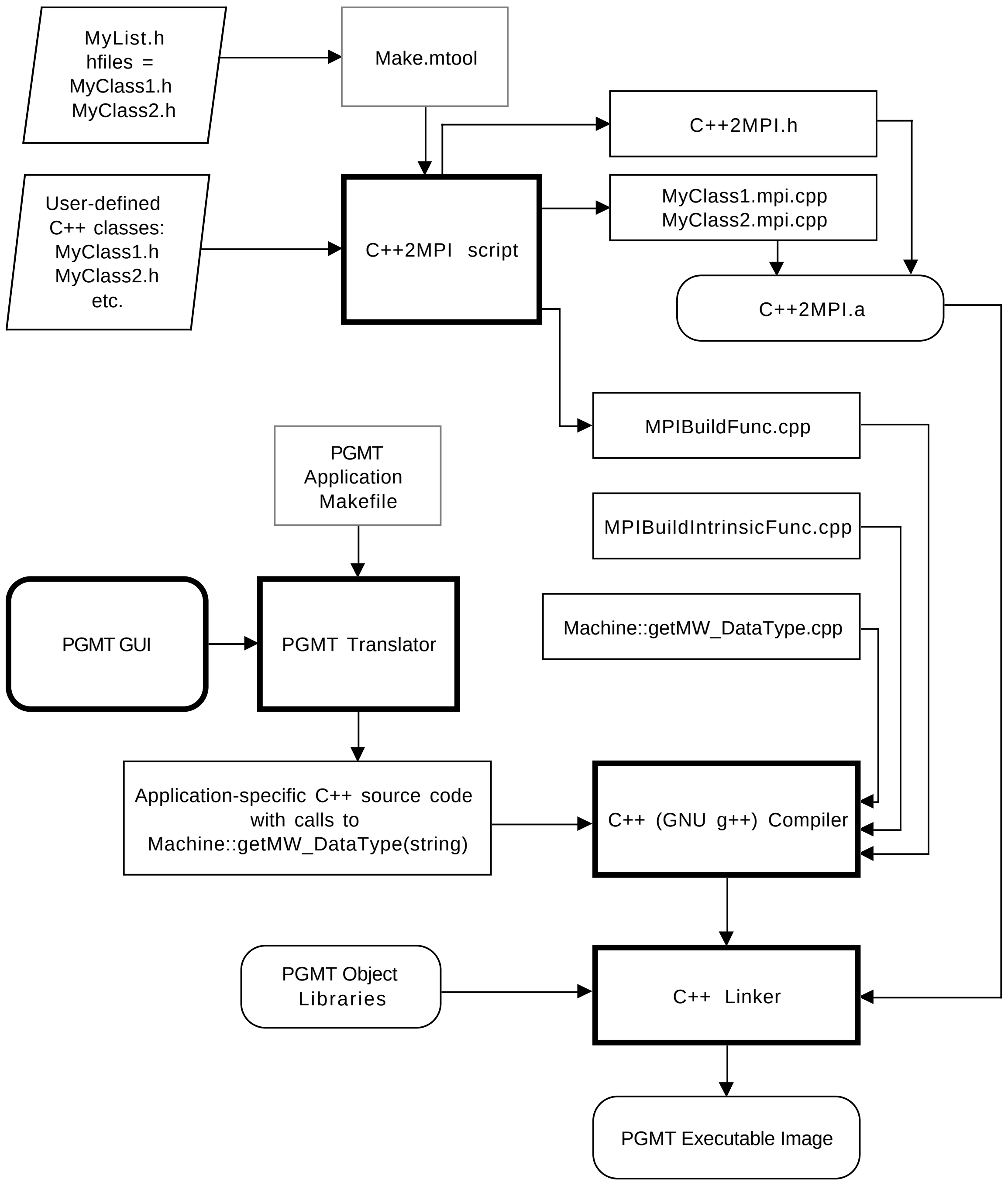

Figure 1: PGMT/C++2MPI Integration

Assume that a user has defined two different classes defining a complex variable. The name of the first class is "complex" and the name of the second class is "fcomplex." The complete C++2MPI output file *MPIBuildFunc.cpp* is:

```
////////////////////////////////////////////////////////////////////////////////////////////////////////
///////////////// MPIBuildFunc.cpp //////////////

#include "C++2MPI_aux.h"

void const GetC++2MPIVector(vector<record> &DFunc)
{
struct record item;
Dfill("complex",  build_complex_MPI_datatype);
Dfill("fcomplex", build_fcomplex_MPI_datatype);
}
////////////////////////////////////////////////////////////////////////////////////////////////////////
```

`Dfill` is a macro. After macro expansion by the C++ preprocessor, the output file `MPIBuildFunc.cpp` is:

```
////////////////////////////////////////////////////////////////////////////////////////////////////////
//  Preprocessor output from MPIBuildFunc.cpp
//  'mpiCC -E MPIBuildFunc.cpp'
//  This demonstrates the Ifill/Dfill macro expansion

//  DFunc.push_back(item) inserts an instance of the structure
//  item into the vector DFunc

void const GetC++2MPIVector(vector<record> &DFunc)
{
struct record item;

extern MPI_Datatype      build_complex_MPI_datatype ();
  item.x =     string( "complex" );
  item.y =     &build_complex_MPI_datatype;
DFunc.push_back(item);

extern MPI_Datatype      build_fcomplex_MPI_datatype ();
  item.x = string( "fcomplex" );
  item.y = &build_fcomplex_MPI_datatype;
DFunc.push_back(item);

}
////////////////////////////////////////////////////////////////////////////////////////////////////////
```

`MPIBuildFunc.cpp` contains a function *GetC++2MPIVector(vector<record> &DFunc).* When this function is called, it returns the vector of structures required to build an STL associative map. The map keys are the class names, and the values returned are the function addresses. The map also includes functions which return the *predefined* MPI datatypes (handles) for the basic C++ arithmetic types. For example, *int* is the key to a function which will the value of MPI_INT. The associative map is accessed via a function call to *getMW_Datatype()* generated by the PGMT GSF-to-C++ translator.

4.2 MPIBuildFunc.cpp A default version of the file `MPIBuildFunc.cpp` is provided with PGMT libraries. The default file is provided so that the Middleware libraries can be compiled and linked without first running C++2MPI.

This is a default file which I have added and committed to CVS. *MPIBuildFunc.cpp* contains the function *void const GetMtoolVector(vector<record> &DFunc).)* The method *Machine::getMW_Datatype()* calls *GetMtoolVector()* the first time *Machine::getMW_Datatype()*itself is called. When called, *GetMtoolVector()* loads the keys and values for the user-defined types into map<>.

The default *MPIBuildFunc.cpp* contains a no-op version of *GetMtoolVector(). Machine::getMW_Datatype()* can therefore be compiled and executed even if the user doesn't execute C++2MPI, an action which would explicitly create *MPIBuildFunc.cpp*. If the user does execute C++2MPI, C++2MPI will create the non-default *MPIBuildFunc.cpp* file, and copy this file to the Middleware directory, overwriting the default file *MPIBuildFunc.cpp*.

V. The PGMT directory structure:

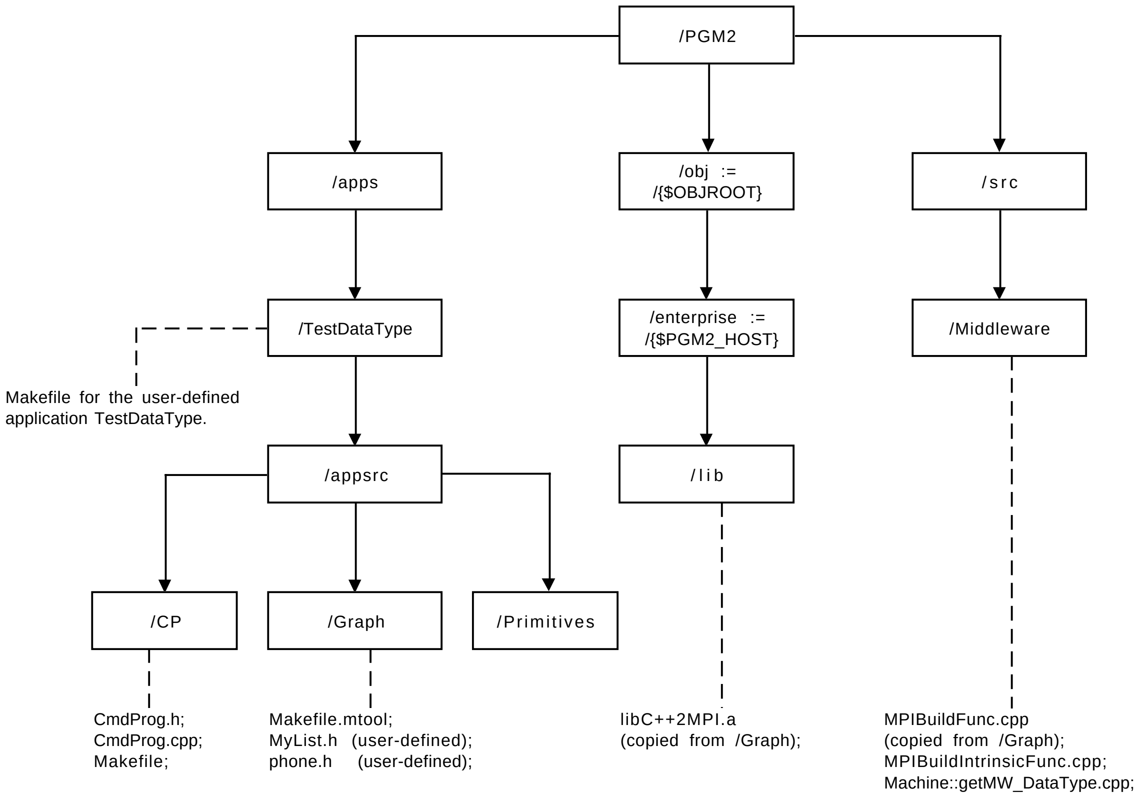
/PGM2
/apps
/obj := /{$OBJROOT}
/src
/TestDataType
/enterprise := /{$PGM2_HOST}
/Middleware
Makefile for the user-defined application TestDataType.
/appsrc
/lib
/CP
/Graph
/Primitives
CmdProg.h; CmdProg.cpp; Makefile;
Makefile.mtool; MyList.h (user-defined); phone.h (user-defined);
libC++2MPI.a (copied from /Graph);
MPIBuildFunc.cpp (copied from /Graph); MPIBuildIntrinsicFunc.cpp; Machine::getMW_DataType.cpp;

Figure 2: PGMT / C++2MPI Directory Structure for a PGMT Application with a User-Defined Type.

Figure 2 illustrates the PGMT directory structure for the example TestDataType. The file phone.h defines the user-defined class for which PGMT requires an MPI datatype. MyList.h contains the list of class files for the user-specified types – in this case, phone.h. Note that the files which the user must provide or modify are in the subdirectory /TestDataType. The files in /obj and /src should not require modification. Directories containing new user applications will be subdirectories of /apps, as is /TestDataType. MPIBuildFunc.cpp is *created* in the directory /graph when C++2MPI is executed, and is then copied (by the Makefile) to /Middleware.

The makefile `Makefile.mtool` in /Graph copies the C++2MPI output file 'MPIBuildFunc.cpp' from /graph to the /Middleware directory, and copies the C++2MPI output archive `libC++2MPI.a` from /graph to /lib.

The functionality of `Makefile.include` in directory /PGM2/apps is also worth summarizing.

> Check to see if the C++2MPI output archive *LibC++2MPI.a* exists. Link with *libC++2MPI.a* if the archive exists; omit this archive from the link library list if *LibC++2MPI.a* does not exist. The archive, if it exists, will be in ${OBJROOT}/${PGM2_HOST}/lib/libC++2MPI.a.

VI. C++2MPI examples:

PGMT provides an additional file `MPIBuildIntrinsicFunc.cpp` defining the handles, function addresses, and function bodies for returning the MPI handles for the predefined MPI datatypes. See Appendix A and B.

Note: if a class is a base-type for a derived class which defines the PGM base-type, a function will be returned for the building the MPI datatype for the base class, but the user does not need to call this function. There will always be a terminal function which the user must call to build a base-type for the desired class; this terminal function may itself call other datatype-building functions to build pre-requisite derived datatypes.

The name of the function for building a derived datatype is the mangled name of the base-type class, or the template signature(s).

Representative input and output files for C++2MPI follow. See the C++2MPI documentation by Michal Iglewski for additional examples.

Example 1: Simple PGM family basetype class. Note use of pragmas to delineate the user-defined class:

```
---------------------------------------------

//  User input file demo1.h
//  file demo1.h

#pragma MPI_START

class demo1{
  public:
    int x1;
    int x2;
    double z;

    demo1() { }
    demo1(int x1_, int x2_, double z_)
    {
      x1 = x1_;
      x2 = x2_;
      z = z_;
    }

    ~demo1() {}
};

#pragma MPI_END

--------------------------------------
```

C++2MPI output prototype file C++2MPI.h. For the sake of brevity, the output file C++2MPI.cpp which contains the function body is not shown.

```
////////// file C++2MPI.h//////////////////////////

#ifndef CPP2MPI
#define CPP2MPI

MPI_Datatype AIT_build_demo1_MPI_datatype();

#endif

----------------------------------------
```

Usage:

```
/*  Declare user-defined datatype demo1_MPI_handle
    and create it by calling AIT_build_demo1_MPI_datatype() */
MPI_Datatype demo1_MPI_handle;
demo1_MPI_handle = AIT_build_demo1_MPI_datatype();
```

Example 2: base class and derived class.

```
// Extract from user input file

class base_v(){...};     // user-defined
class derived_v(){...);  // user-defined

--------------------------------------------------------
/*  prototypes generated by C++2MPI; function bodies are not
shown.  */
MPI_Datatype AIT_build_base_v_MPI_datatype();    //C++2MPI out
MPI_Datatype AIT_build_derived_v_MPI_datatype();// C++2MPI out

---------------------------------------------------------

// Usage:

MPI_Datatype derived_v_MPI_handle;
derived_v_MPI_handle = AIT_build_derived_v_MPI_datatype();

/*  Note that the function AIT_build_derived_v_MPI_datatype()
calls the function AIT_build_base_v_MPI_datatype()internally.
The base class function does not have to be called from within
the PGMT C++ source code.             */

----------------------------------------------------------
```

Example 3: templated class for user-defined PGM family basetype. Each permissible combination of input parameters must be type-defined by the user.

```
----------------------------------------------
```

User-defined input file:

```
//file Nemo.h
#pragma MPI_START

template <class T> class Nemo {public: T x; T y;};

typedef Nemo<int> NI;
typedef Nemo<float> NF;

----------------------------------------------
```

C++2MPI output prototype file C++2MPI.h. Function bodies in C++2MPI.cpp are not shown.

```
///////////////// file C++2MPI.h/////////////////////////

#ifndef CPP2MPI
#define CPP2MPI

MPI_Datatype AIT_build_Nemo_int_MPI_datatype();
MPI_Datatype AIT_build_NI_MPI_datatype();
MPI_Datatype AIT_build_Nemo_float_MPI_datatype();
MPI_Datatype AIT_build_NF_MPI_datatype();

#endif
//  end C++2MPI.h

---------------------------------------------------------
```

Notice that C++2MPI builds a prototype for the type-defined variables NF and NI, as well as for Nemo<float> and Nemo<int>.

The function `Datatype AIT_build_NI_MPI_datatype()` is actually a wrapper for a function call to `build_Nemo_int_MPI_datatype() i.e.`

```
//  code fragment from the C++2MPI output file C++2MPI.cpp:

//  Build an MPI Datatype for the type 'NI'
MPI_Datatype AIT_build_NI_MPI_datatype()
{
 return AIT_build_Nemo_int_MPI_datatype();
} // end AIT_build_NI_MPI_datatype function

-----------------------------------------------------------
```

Usage:

```
MPI_Datatype NI_MPI_handle;
derived_v_MPI_handle = AIT_build_NI_MPI_datatype();
```

III. Summary of the C++2MPI name-mangling algorithm

I am providing this information for reference. PGMT/C++2MPI users do not have to be concerned with this algorithm.

i. For a simple (non-templated) class, the class name is inserted into the function name.

```
class MyClass();  // user-defined base type MyClass
//  Prototype for MPI datatype building function
MPI_Datatype AIT_build_MyClass_MPI_datatype();
```

ii. If the user's class is a derived datatype, a build function must be provided for the base type, as well as the derived type. For example, assume there is user-specified base class **base_v()**, and a user-specified derived class **derived_v()**. The prototypes for the build functions are:

```
MPI_Datatype AIT_build_base_v_MPI_datatype();
MPI_Datatype AIT_build_derived_v_MPI_datatype();
```

iii. Nested classes.

For nested, but non-templated, classes there will be a build function for each class. The name of the function reflects the name of the class. Consider a class **C1** which contains classes **local** and **C2**; **C2** also declares an object of type **Local**.

The C++2MPI output file is:

```
////////////////////// file C++2MPI.h ////////////////////////

#ifndef CPP2MPI
#define CPP2MPI

MPI_Datatype AIT_build_Local_MPI_datatype();
MPI_Datatype AIT_build_C2_MPI_datatype();
MPI_Datatype AIT_build_C1_MPI_datatype();

#endif
```

iv. Templated basetypes. Syntax is introduced for templated arguments to other templates.

```
//user input file fif.h:

#pragma MPI_START

template <class T3> class T2 {public: T3 y; };
template <class T2> class T1 {public: T2 x; };

//  Instantiate T1 as a function of class T2<float>
typedef T1< T2<float> > fif;

//  End fif.h

---------------------------------------------------------

//  C++2MPI output file C++2MPI.h:

#ifndef CPP2MPI
#define CPP2MPI

MPI_Datatype AIT_build_T2_float_MPI_datatype();
/*  Note use of the word class to delineate the 'inner' template
instantiation.  */
MPI_Datatype AIT_build_T1_class_T2_float_MPI_datatype();
MPI_Datatype AIT_build_fif_MPI_datatype();

#endif

/*  In the calling program, only AIT_build_fif_MPI_datatype()
must be called. */
-----------------------------------------------------------
```